20. Applications

Created: April 1, 2003
Updated: October 10, 2003

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Most of the applications discussed in this chapter are built on a regular basis, at least once a day from the latest sources, and if you are in NCBI, then you can find the latest version in directory: *$NCBI/c++/Release/bin/* (or *$NCBI/c++/Debug/bin/*).

Chapter Outline

The following is an outline of the topics presented in this chapter:

- DATATOOL: Code Generation and Data Serialization Utility
  - Invocation
    - Main arguments
    - Code generation arguments
  - Definition file
    - Common definitions
    - Definitions which affect specific types
      - INTEGER, REAL, BOOLEAN, NULL
      - ENUMERATED
      - OCTET STRING
      - SEQUENCE OF, SET OF

            - SEQUENCE, SET
            - CHOICE
        - Examples
    - Generated code
        - Normalized name
        - ENUMERATED types
        - SEQUENCE and SET code
        - CHOICE code
        - Other types code
    - Class diagrams
        - Specification analysis
            - ASN.1 specification analysis
            - DTD specification analysis
        - Data types
        - Data values
        - Type strings
        - Code generation
- Load-Balancing Service Mapping Daemon (LBSMD)
    - Overview
    - LBSMD in Details
    - Example of Configuration File
    - Load-Balancing Service Mapping Client (LBSMC)
    - Server Penalizer

- Network Service Mapper/Dispatcher (DISPD.CGI)
  - Overview
  - Protocol Description
    - Client Request to DISPD.CGI
    - DISPD.CGI Response to Client
  - Communication Schemes
  - Server Launcher (NCBID.CGI)
- NCBI Firewall Daemon (FWDaemon)
  - Using FD to connect from behind a "regular" firewall
  - Using FD to connect from behind a "non-transparent" firewall
  - Troubleshooting
- NCBI Genome Workbench
  - Design Goals
  - Design

DATATOOL: Code Generation and Data Serialization Utility

The datatool is located at*c++/src/serial/datatool* and can perform the following:

1. Generate C++ data storage classes based on ASN.1 or DTD specification to be used with NCBI data serialization streams.
2. Convert ASN.1 specification into DTD and vice versa.
3. Convert data between ASN.1 and XML formats.

NOTE: Since ASN.1 and DTD are, in general, incompatible, the last two functions are supported only partially.

The following additional topics are discussed in subsections:

- Invocation
- Definition file
- Generated code
- Class diagrams

Invocation

The following topics are discussed in this section:

- Main arguments
- Code generation arguments

Main arguments

See Table 1.

Table 1. Main Arguments

Argument	Effect	Comments
-h	display the `DATATOOL` arguments	Ignores other arguments
-m <file>	ASN.1 or DTD module file(s)	Required argument
-M <file>	external module file(s)	Is used for IMPORT type resolution
-i	ignore unresolved types	Is used for IMPORT type resolution
-f <file>	write ASN.1 module file	
-fx <file>	write DTD module file	"-fx m" writes modular DTD file
-fxs <file>	write XML Schema file	
-dn <string>	DTD module name in XML header	no extension
-v <file>	read value in ASN.1 text format	
-vx <file>	read value in XML format	
-F	read value completely into memory	
-p <file>	write value in ASN.1 text format	
-px <file>	write value in XML format	
-d <file>	read value in ASN.1 binary format	*-t* argument required
-t <type>	binary value type	see *-d* argument
-e <file>	write value in ASN.1 binary format	
-sxo	no scope prefixes in XML output	
-sxi	no scope prefixes in XML input	
-logfile <File_Out>	File to which the program log should be redirected	
conffile <File_In>	Program's configuration (registry) data file	
-version	Print version number	Ignores other arguments

Code generation arguments

See Table 2.

Table 2. Code Generation Arguments

Argument	Effect	Comments
-od <file>	C++ code definition file	see Definition file
-odi	ignore absent code definition file	
-odw	issue a warning about absent code definition file	
-oA	generate C++ files for all types	only types from the main module are used (see *-m* and *-mx* arguments)

Argument	Effect	Comments
-ot <types>	generate C++ files for listed types	only types from the main module are used (see *-m* and *-mx* arguments)
-ox <types>	exclude types from generation	
-oX	turn off recursive type generation	
-of <file>	write the list of generated C++ files	
-oc <file>	write combining C++ files	
-on <string>	default namespace	
-opm <dir>	directory for searching source modules	
-oph <dir>	directory for generated *.hpp files	
-opc <dir>	directory for generated *.cpp files	
-or <prefix>	add prefix to generated file names	
-orq	use quoted syntax form for generated include files	
-ors	add source file dir to generated file names	
-orm	add module name to generated file names	
-orA	combine all -or* prefixes	
-ocvs	create ".cvsignore" files	
-oR <dir>	set -op* and -or* arguments for NCBI directory tree	
-lax_syntax	allow non-standard ASN.1 syntax accepted by asntool	
-oex <export>	add storage-class modifier to generated classes	can be overriden by *[-]._export* in the definition file

Definition file

It is possible to tune up the C++ code generation by using a definition file, which could be specified in *-od* argument. Definition file utilizes the generic NCBI configuration format also used in NCBI application's configuration (*.ini) files.

DATATOOL looks for code generation parameters in several sections of the file in the following order:

1. *[ModuleName.TypeName]*
2. *[TypeName]*
3. *[ModuleName]*
4. *[-]*

Prefix of the parameter name in section is determined from the location of an element in the data format specification (ASN.1 or DTD). For the root element prefix is empty. For an element of type *SET OF* or *SEQUENCE OF* - add *E.* to prefix. For an element of type *SET*, *SEQUENCE* or *CHOICE* - add the element name and dot (".") to prefix.

The following additional topics are discussed in this section:

- Common definitions
- Definitions which affect specific types
- Examples

Common definitions

Some definitions refer to the generated class as a whole: *_file* Defines the base file name for the generated C++ class.

> For example, the following definitions: *[ModuleName.TypeName]_file=AnotherName* or *[TypeName]_file=AnotherName* would put the class ***CTypeName*** in files with the base name *AnotherName*. While these two: *[ModuleName]_file=AnotherName* or *[-]_file=AnotherName* put **all** the generated classes into a single file with the base name *AnotherName*.

_dir Subdirectory in which the generated C++ files will be stored (in case *_file* not specified). *_class* The name of the generated class (if *_class*=- is specified, then no code is generated for this type).

> For example, the following definitions: *[ModuleName.TypeName]_class=AnotherName* or *[TypeName]_class=AnotherName* would cause the class generated for the type *TypeName* to be named ***CAnotherName***. While these two: *[ModuleName]_class=AnotherName* or *[-]_class=AnotherName* would result in **all** the generated classes having the same name ***CAnotherName*** (which is probably not what you want).

_namespace The namespace in which the generated class (or classes) will be placed. *_parent_class* The name of the base class from which the generated C++ class is derived. *_parent_type* Derive the generated C++ class from the class, which corresponds to the specified type (in case *_parent_class* is not specified).

It is also possible to specify a storage-class modifier, which is required on Microsoft Windows to export/import generated classes from/to a DLL. This setting affects all generated classes in a module. Appropriate section of the definition file should look like this:

```
[-]_export = EXPORT_SPECIFIER
```

Since this modifier could also be specified in the command line, DATATOOL code generator uses the following rules to choose the proper one:

1. If no *-oex* flag is given in the command line, then no modifier is added at all.
2. If *-oex ""* (that is, an empty modifier) is specified in the command line, then the modifier from the definition file will be used.
3. The command line parameter in the form *-oex FOOBAR* will cause the generated classes to have *FOOBAR* storage-class modifier, unless another one is specified in the definition file. The modifier from the definition file always takes precedence.

Definitions which affect specific types

The following additional topics are discussed in this section:

- INTEGER, REAL, BOOLEAN, NULL
- ENUMERATED
- OCTET STRING
- SEQUENCE OF, SET OF
- SEQUENCE, SET
- CHOICE

INTEGER, REAL, BOOLEAN, NULL

_type C++ type: int, short, unsigned, long etc.

ENUMERATED

_type C++ type: int, short, unsigned, long etc. *_prefix* Prefix for enum values' names. Default is "e".

OCTET STRING

_char Vector element type: char, unsigned char or signed char.

SEQUENCE OF, SET OF

_type STL container type: list, vector, set, or multiset.

SEQUENCE, SET

memberName._delay Mark the specified member for delayed reading.

CHOICE

_virtual_choice If non-empty, do not generate a special class for choice. Rather make the choice class as parent one of all its variants. *variantName._delay* Mark the specified variant for delayed reading.

Examples

If we have the following ASN.1 specification:

```
Date ::= CHOICE {
    str VisibleString,
    std Date-std
}
Date-std ::= SEQUENCE {
    year INTEGER,
    month INTEGER OPTIONAL
}
Dates ::= SEQUENCE OF Date
Int-fuzz ::= CHOICE {
    p-m INTEGER,
    range SEQUENCE {
        max INTEGER,
        min INTEGER
    },
    pct INTEGER,
    lim ENUMERATED {
        unk (0),
        gt (1),
        lt (2),
        tr (3),
        tl (4),
        circle (5),
        other (255)
    },
    alt SET OF INTEGER }
```

Then the following definitions:

[Date]str._type = string

would affect the generation of str member of the Date structure.

[Dates]E._pointer = true

would affect the generation of elements of the Dates container.

[Int-fuzz]range. min._type = long

would affect the generation of the min member of the range member of the Int-fuzz structure.

[Int-fuzz]alt.E._type = long

would affect the generation of elements of the alt member of the Int-fuzz structure.

Generated code

The following additional topics are discussed in this section:

- Normalized name
- ENUMERATED types
- SEQUENCE and SET code
- CHOICE code
- Other types code

Normalized name

Everywhere in generated code we use so-called *NormalizedName* which is produced from an ASN.1 type name by replacing all minuses ("-") with underscores ("_") and making first letter capital.

ENUMERATED types

By default, for every ENUMERATED type `DATATOOL` will produce a C++ enum type with the name *ENormalizedName*.

SEQUENCE and SET code

CHOICE code

Other types code

Class diagrams

The following topics are discussed in this section:

- Specification analysis
- Data types
- Data values
- Type strings
- Code generation

Specification analysis

The following topics are discussed in this section:

- ASN.1 specification analysis
- DTD specification analysis

ASN.1 specification analysis

See Figure 1.

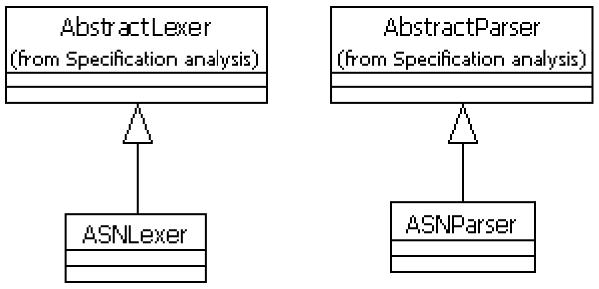

Figure 1: ASN.1 specification analysis

DTD specification analysis

See Figure 2.

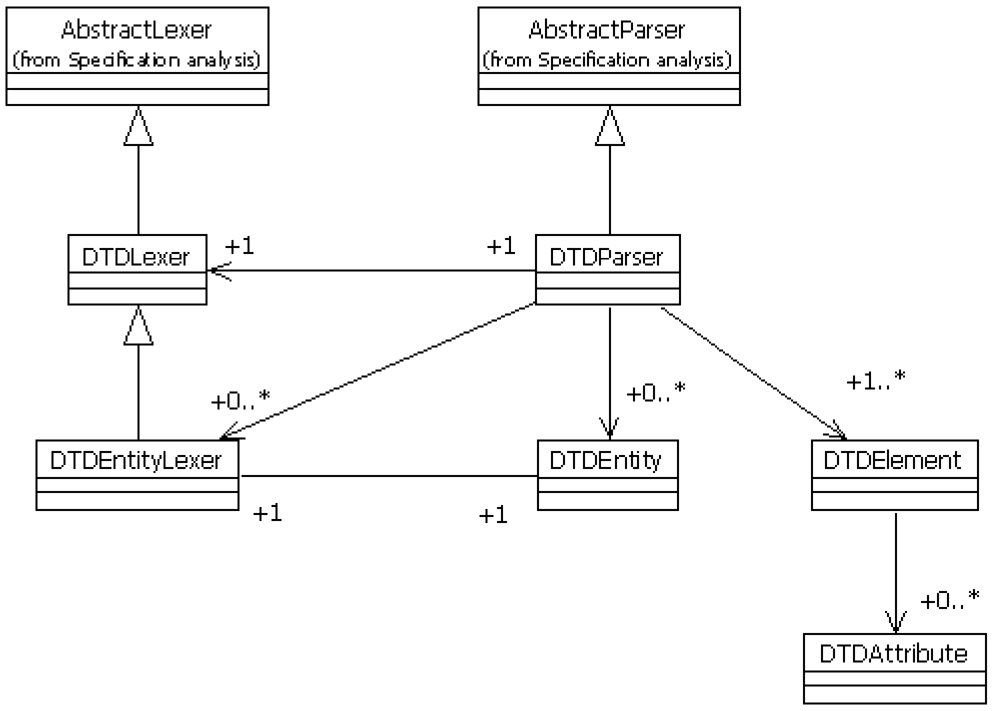

Figure 2: DTD specification analysis

Data types

See Figure 3.

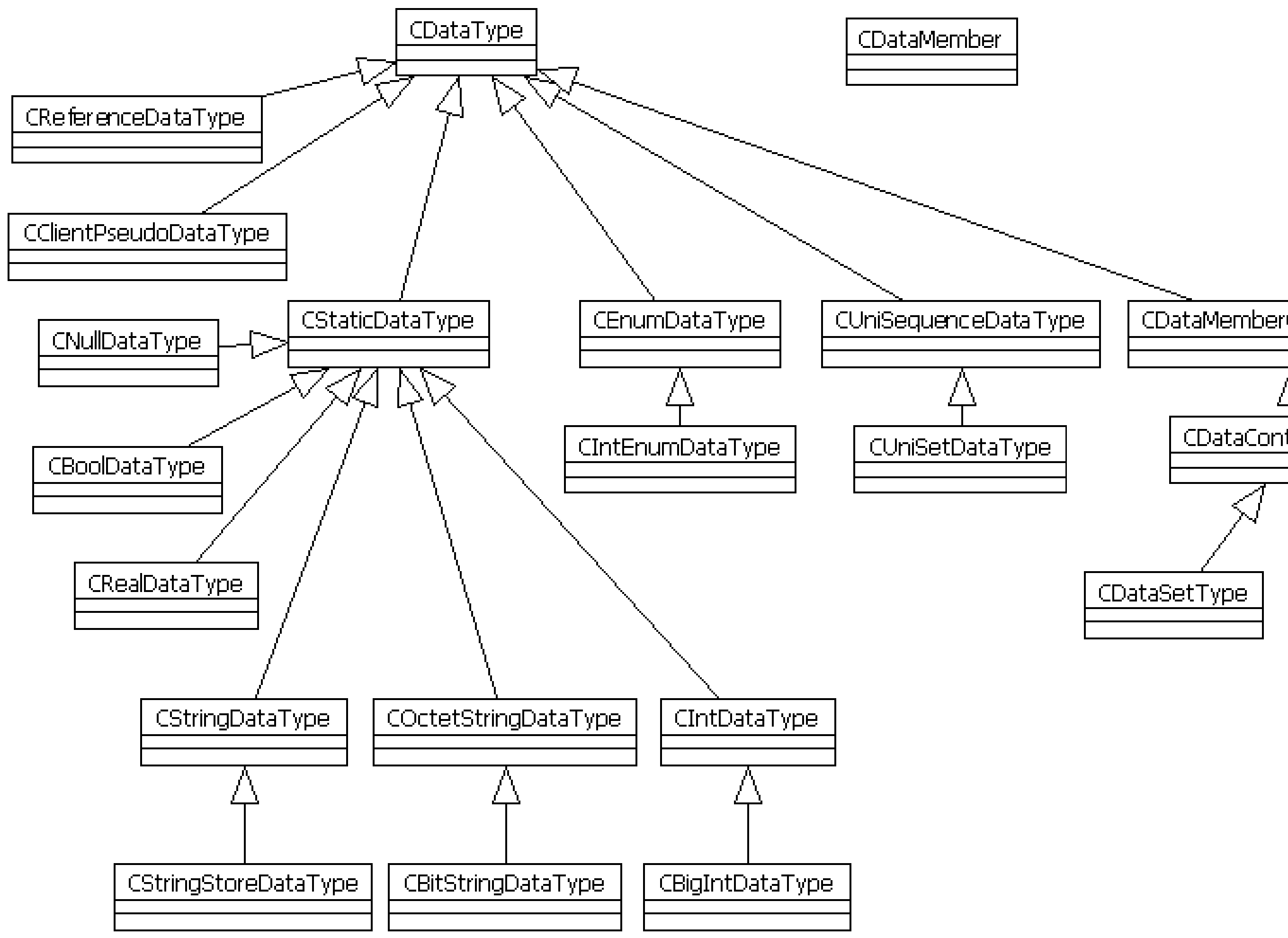

Figure 3: Data types

Data values

See Figure 4.

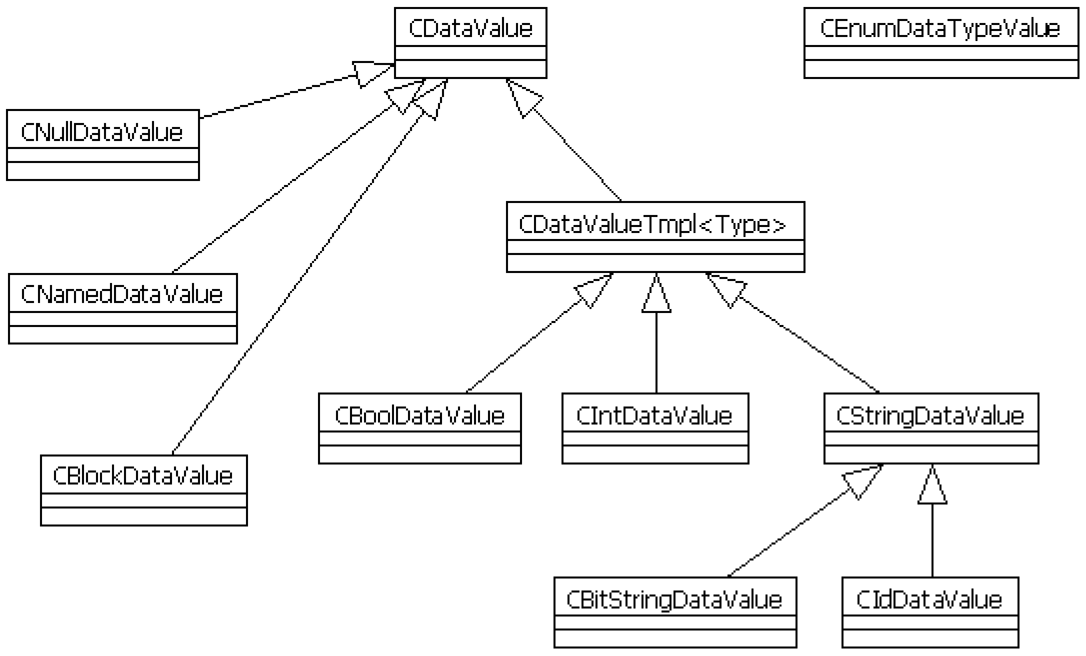

Figure 4: Data values

Type strings

See Figure 5.

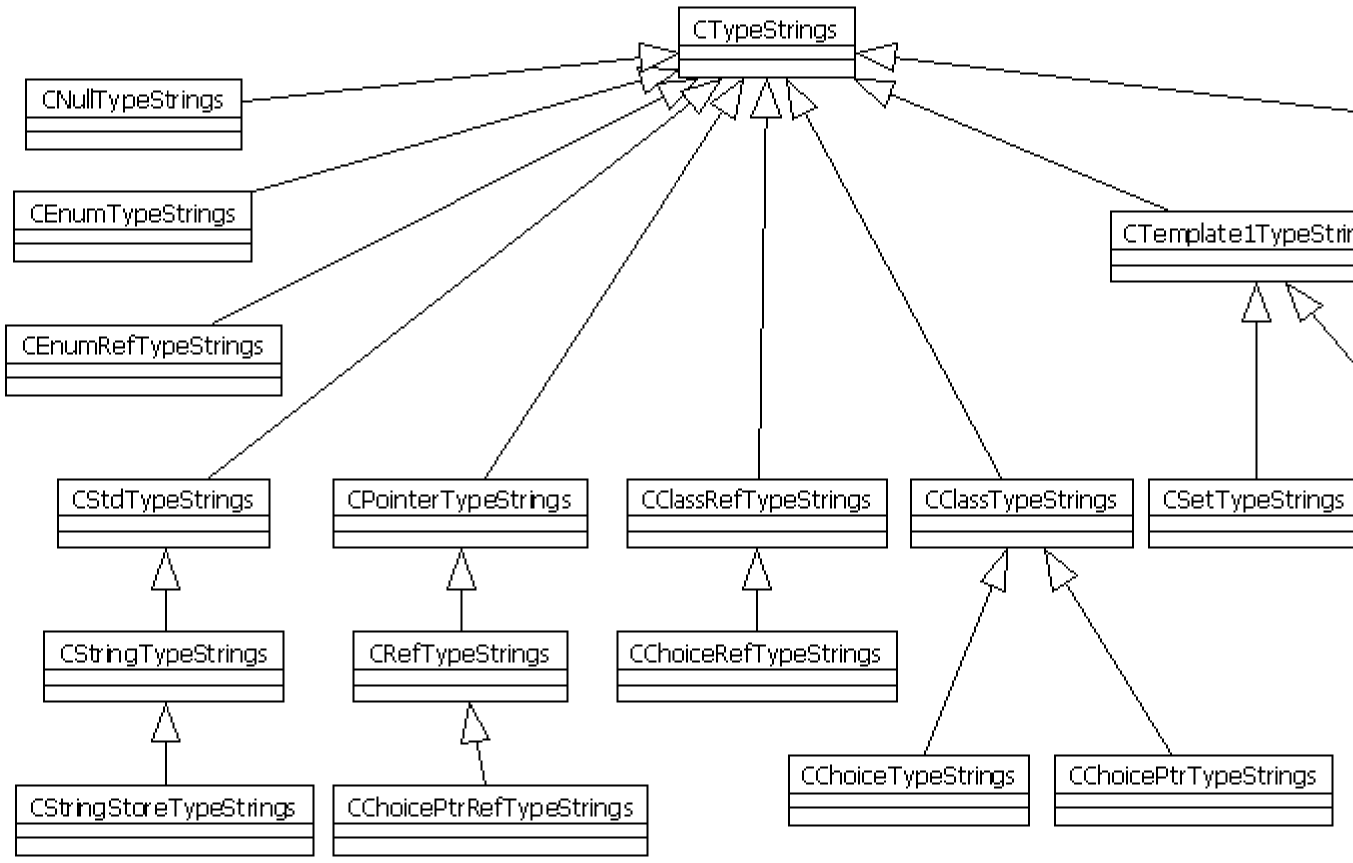

Figure 5: Type strings

Code generation

See Figure 6.

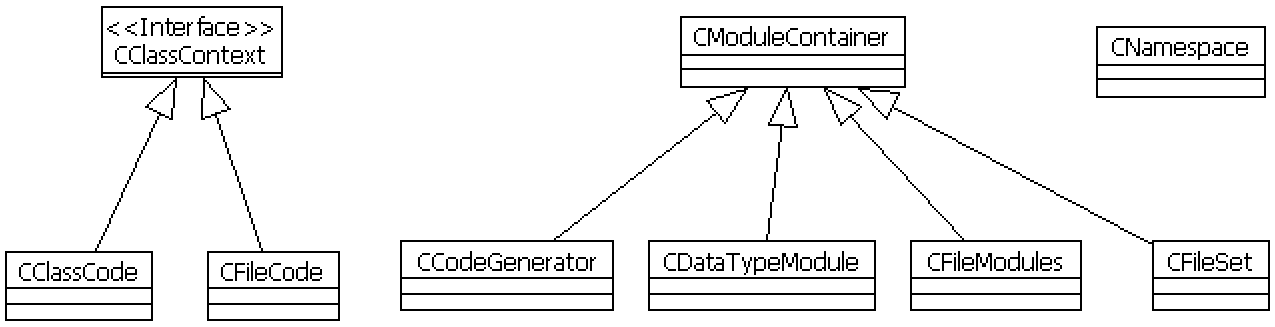

Figure 6: Code generation

Load-Balancing Service Mapping Daemon (`LBSMD`)

Note: Due to security issues, not all links in the public version of this file could be accessible by outside NCBI users. Unrestricted version of this document is available to inside NCBI users at: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/CPP_DOC/tools/dispatcher/LBSMD.html.

The following additional topics are discussed in this section:

- Overview
- LBSMD in Details
- Example of Configuration File
- Load-Balancing Service Mapping Client (LBSMC)
- Server Penalizer

Overview

In general, `LBSMD` should be run at NCBI site on every host that carries either public or private servers implementing NCBI service(s) (see however discussion about *static* servers). The services include CGI programs or standalone servers to access NCBI data. Each service has a name assigned to it, `TaxServer` for instance. The name uniquely identifies the service, and - **most importantly** - the protocol used for data exchange, i.e. any client connecting to service `TaxServer` knows how to communicate with that service, whereas the service can be implemented in various ways on hosts across NCBI. That is, the service could be implemented as a standalone server on a host *X*, and as a CGI program on the same or another host *Y*. (Note however, that on a certain host there should **not** be more than one server of the same type for the same service.) Moreover, the same server can implement several services, that is services with different names. For example, one service (like `Entrez2`) can accept binary data only, and another one (say `Entrez2Text`) can accept data only in text format. For the server, the distinction between those two could be in this case made by using a content type specifier.

The goal of `LBSMD` is to maintain a table of all services available at NCBI at the moment. In addition, `LBSMD` keeps track of servers, which are found non-functional (dead servers), and it is also responsible for propagating trouble reports, obtained back from the applications about their experience with advertized servers (e.g. an advertized server is not technically dead, but generates sort of garbage when accessed). Further in this document the latter kind of feedback is called a *penalty*.

The principle of load-balancing is simple: each server implementing a service is assigned a (calculated) rate. The higher the rate the better the chance for that the server to be chosen. Note that load-balancing is thus almost never deterministic (see however, discussion of negative local bonus coefficient).

It is not obvious that the load-balancing daemon does not calculate the rates of servers. Instead, its primary goal is to collect all necessary information for that. The server calculates two parameters, normal host status and so called BLAST host status (based on the instant load of the system). These parameters are then used to calculate the rate of all (non-*static*) servers on that host as shown in *connect/ncbi_lbsm.c* (which is a part of service mapping API, not `LBSMD`).

`LBSMD` in Details

`LBSMD` is configured by means of a configuration file (default name *servrc.cfg*), which is also shared with NCBI CGI engine, `NCBID.CGI`. Each line in the file either defines a service, or is a part of the host environment, or is an include directive, or, finally, is an empty line (the one entirely blank or containing a comment only). Empty lines are all ignored in the file. Any single configuration line can be split into several physical lines by inserting backslash symbols (\) before the line breaks. A comment is introduced by the hash symbol (#).

A configuration line of the form

```
name=value
```

places itself in the host environment. The host environment can be accessed by clients when they make the service name resolution. Initially, the host environment was designed to help the client know about limitations/options, which the host has, and based on this additional information the client can make a decision whether the server (despite that it is implementing the service) is suitable for carrying out client's request. For example, the host environment can give the client an idea about what are the databases available on the host. The host environment is not interpreted or used in any way both by the daemon and by the load-balancing algorithm, except for that `name` must be a valid identifier. The *value* may be practically anything, even empty. It is left solely to the client to parse the environment and to look for the information in interest. The host environment can be obtained from the service iterator via call to ***SERV_GetNextInfoEx()***, which is documented in service mapping API. **Note**: White characters surrounding `name` are not preserved, but they are preserved in *value*, i.e. when appear after the assignment sign.

A configuration line of the form

```
%include filename
```

causes the named file `filename` to be read at this point of the surrounding configuration file. The daemon always assumes that relative file names (those which names do not start with the slash character (/)) are looked for with the daemon startup directory as a base. This is true for any level of nesting.

Once started the daemon first assignes the top configuration file as */var/etc/lbsm/servrc.cfg*, then tries to read it. If the file is not found (or is not readable), the daemon looks for the configuration file *servrc.cfg* in the directory, from which it has been started. If found, that file is assigned as the top configuration file. This fallback machanism is not used when the configuration file name is explicitly stated in the command line. When running, the daemon periodically checks the top configuration file and all its descendants and reloads (discards) their contents if some of the files have been either updated, (re-)moved, or added.

A `service` is defined by a line of the form

```
service_name [check_time] server_descriptor [| launcher_info ]
```

- `service_name` names the service, for instance `TaxServer`.
- `[check_time]` is an optional parameter (if omitted, the surrounding square brackets must go, too), which specifies the number of seconds for periodic checkups. For example, *[120]* means to check the server once every 2 minutes. Note the square brackets - they are required. Checkups can only be done for servers running locally, that is specifying a check timeout for any server located outside the local machine causes a warning printed and the timeout ignored.
- `server_descriptor` specifies address of the server and supplies additional information, which is described in details later in this document. An example of the `server_descriptor`:

  ```
  STANDALONE somehost:1234 R=3000 L=yes S=yes B=-20
  ```

- `launcher_info` is basically a command line preceded by a pipe symbol (`|`), delimiting from `server_descriptor`. It is only required for servers of type`NCBID`, which are configured on the local host.

`Server_descriptor`, also detailed in *connect/ncbi_server_info.h*, consists of the following fields: `server_type [host][:port] [arguments] [flags]` where:

- `server_type` is one of the following keywords (see also here):
  - `NCBID` for servers launched by `ncbid.cgi`;
  - ***STANDALONE*** for standalone servers listening on dedicated ports;
  - ***HTTP_GET*** for servers, which are the CGI programs accepting only `GET` request method;

  - ***HTTP_POST*** for servers, which are the CGI programs accepting only `POST` request method;
  - ***HTTP_POST*** for servers, which are the CGI programs accepting both of either `GET` or `POST` request methods;
  - ***DNS*** for introduction of a name (fake service), which can be later used in load-balancing domain name resolution;
  - ***NAMEHOLD*** for declaration of service names that cannot be defined in any other configuration files except for the current configuration file. **Note**: ***FIREWALL*** server specification may not be used in configuration file (i.e. may neither be declared as services nor as service name holders).

- Both `host` and *port* parameters are optional. Defaults are local host and port 80 except for ***STANDALONE*** and ***DNS*** servers, which **do not have** a default port value. If `host`**is specified** (by either of the following: keyword *localhost*, localhost IP address 127.0.0.1, real host name or IP address) then the described server is not a subject for variable load-balancing, but is a so-called *static* server. Such a server always has a constant rate, independent of any host load.
- `arguments` are required for HTTP* servers, and must specify local part of URL of the CGI program and optionally parameters, like: `/somepath/somecgi.cgi?param1&param2=value2&param3=value3`If no parameters are to be supplied then the question mark (?) must be omitted, too. For `NCBID` servers, `arguments` are parameters to pass to the server and are formed as arguments for CGI programs, i.e. `param1&param2&param3=value` As a special rule, '' (two single quotes) may be used to denote an empty argument for `NCBID` server. ***STANDALONE*** and ***DNS*** servers do **not** take any `arguments`.
- `flags` can come in no specific order (but no more than one instance of a flag is allowed), and essentially are the optional modifiers of values used by default. The following flags are recognized:

  - load calculation keyword:

    - *Blast* to use special algorithm for rate calculation acceptable for BLAST applications. The algorithm uses instant values of the host load, and thus is less conservative and more reactive than the ordinary one;
    - *Regular* to use an ordinary rate calculation (default, and the only load calculation option allowed for*static* servers).

- base rate:
  - *R=value* sets base server reachability rate (as a floating point number), default is 1000. Any negative value makes the server unreachable, and a value of 0 uses the default. The maximal allowed base rate is 100000.
- locality markers:
  - *L={yes|no}* sets (if *yes*) the server to be *local only*. Default is *no*. Service mapping API returns *local only* servers in the case of mapping with the use of `LBSMD` running on the same - local - host (direct mapping), or if the dispatching (indirect mapping) occurs within the NCBI Intranet. Otherwise, if service mapping occurs using non-local network (certainly indirectly, via exchange with`dispd.cgi`) then *local only* servers are not seen.
  - *P={yes|no}* sets (if *yes*) the server to be *private*. Default is *no*. Private servers are not seen by the outside users (exactly like *local* servers), but in addition these servers are not seen from the NCBI Intranet if requested from a host, which is different from one where the private server runs. This flag cannot be used for ***DNS*** servers.

  Note: If necessary, both *L* and *P* markers can be combined in a particular service definition.
- stateful server:
  - *S={yes|no}* sets (if *yes*) that the server can only accept dedicated connections, and is not capable of doing *stateless* (HTTP-like) data exchange. Such a server can only be accessed by creating a stream connection to it, and the entire data transfer is to happen while the connection is alive. Default is *no*. This flag **cannot** be specified for HTTP* servers, and is silently ignored for ***DNS*** servers. **Note**: Due to mutual incompatibility, Web-browsers *cannot* directly connect to *stateful* servers of any type.
- content type specifier:
  - *C=type/subtype* specifies what is the data type that the server expects. This content type is to be used by a client, which wants to connect to the server. There is no default value for this flag, and the flag cannot be used for ***DNS*** servers.

- local bonus coefficient:

    - *B=value*. The interpretation of this flag depends on the sign of *value*. When positive, *value* specifies a multiplier, which applies to the server rate when the server is locally run. When zero (default), the slight default rate increase takes effect for locally run servers. When *value* is negative, the locally run server overrides those even having higher rates and preceding the local one, but only if the local server has a rate, which is not in the percentile (expressed by the absolute *value*) of average remaining rate of all other servers, implementing the service. For example, a server on a *local* host *Z* configured with *B=-5* (and currently having rate of 100) will be chosen by the *local* load-balancer even if there were servers on hosts *X* and *Y* having rates 1000 and 2000, respectively, because 100 is more than 5% of (1000 + 2000)/2 = 1500, which is 75. **Note** again that for this coefficient to play, **both** server and mapping must be on the same host.

- backup quorum:

    - *Q={value|yes|no}*. Server specifications having this flag set define so-called backup configuration, and must have host distinct from the local host. Also, this flag cannot apply to ***NAMEHOLD*** specifications. In a simple case *Q=yes* defines a backup entry, which gets activated when number of dynamic entries (which came from other hosts) for particular service becomes less than the number of the service's backup lines defined in the entire configuration (i.e. counting all files). The order in which backup entries are then chosen is defined by their order of appearance in configuration. In a more complex case, *Q=value* can be provided with the value corresponding to the minimal requirement on the number of active dynamic entries, regardless of the number of backup entries in configuraiton. When this quorum is met, no backup entries are being activated (or will be deactivated and put back pending). **Note**: If several configuration lines for a partical service have *Q=value* flag then the quorum is the minimal value among specified. *Q=no* or *Q=0* defines an active service entry (as if *Q* flag were not specified at all).

Server descriptors of type ***NAMEHOLD*** are special. As `arguments` they have only server type keyword. Namehold specification tells to the daemon that service of this name and type is not to be defined later in any configuration file except for the current. Also, if host is specified then this protection only works for the service name on the particular host. Port number is ignored (if specified). **Note**: it is recommended to always put a dummy port number (like :0) in namehold specifications to avoid ambiguities with treating server type as a host name. The followin example disables `TestService` of type ***DNS*** to be defined in all other configuration files included later, and `TestService2` to be defined as a `NCBID` service on host *foo*:

```
TestService  NAMEHOLD    :0 DNS
TestService2 NAMEHOLD foo:0 NCBID
```

The main loop of the daemon comprises periodic checks of configuration file and reloads of the configuration if necessary, checks and processings of incoming messages from load-balancing daemons running on other hosts, generation and brodacast of the messages to other hosts about the load of the system and configured services. Also, the daemon periodically checks whether the configured servers are alive, trying to connect to them, and then to immediately disconnect, without sending/receiving any data. This way the daemon is only able to check if the network port is working. Further server accessibility information could be sent back to the daemon in the form of a penalty from applications that actually use the server.

The daemon can be configured by a dozen of switches, which modify certain parameters from their default values. To see all command-line switches, switch *-h* may be used, or NCBI Intranet users **only** can click here. Also, daemon reacts on `HUP` signal to reload its configuration, and both `INT` and `TERM` signals to gracefully quit. Despite very high stability, usually the daemon is forcedly started from *crontab* each every few minutes on all production hosts to ensure that the daemon is always running. This technique is safe, because no more than one instance of the daemon is permitted on a certain host, and any attempt to start more than one is rejected.

`LBSMD` can generate output to the logfile, which can be limited in size to prevent disk from flooding with messages. NCBI Intranet users **only**can take a look at (no more than 100) recent lines of the logfile on host *ray*. Signal `USR1` can be used to toggle verbosity level between less verbose (default) and more verbose (when every warning generated is stored) modes.

Logfile size can be controlled by *-s* switch. By default *-s 0* is the active flag, which means to create (if necessary) and to append messages to the logfile with no limitation on the file size whatsoever. *-s -1* instructs indefinite appending to the logfile, which has however to exist. Otherwise, log messages are not stored. *-s positive_number* restricts to create (if necessary) and to append to the logfile until the file reaches the specified size in kilobytes. After that, the message logging is suspended, and subsequent messages are discarded. Note that the limiting file size is only approximate, and sometimes the logfile can grow slightly bigger. The daemon keeps track of logfiles and leaves the final logging message either when switching from one file to another in case of the file has been moved or removed or when the file size has reached its limit.

Example of Configuration File

Below is an excerpt from sample configuration file, which declares some of the host environment, advertizes one standalone stateful server, and one `NCBID` service, both implementing the same service, and one local HTTP service.

```
#
#    This is a configuration file of new NCBI service dispatcher
#

# This goes to host environment (rather practical)
db_list=a.db b.db c.db
# And this does too (rather illustrative)
my_entry="some very important value as a quoted string"

# Standalone server listening on port 9999
TestService STANDALONE :9999 S=yes R=2000

# NCBID server (Note empty quotes to prevent treating of Regular as
# an argument. Also note bar (|) followed by path and parameters of
# the executable.)
TestService NCBID '' Regular |
   /home/webuser/TestService/a.out "parameter 1"

# Local HTTP service w/o checks (note [0])
LocalService [0] HTTP /LocalService.cgi?param=somevalue L=yes

# Static server - check it each 200 seconds
StaticService [200] STANDALONE localhost:8888

# Static server from another host where LBSMD may even not be running
# (no checks at all or just [0] allowed for non-local static servers)
ForeignService [0] STANDALONE foreignhost:7777 R=300

# Backup entry for DNS service test_lb - will be activated if no dynamic
# entry would be found in the table of services
test_lb DNS otherhost Q=yes

# Hold service name PrivateService of type HTTP from being defined in
# included configuration files
PrivateService NAMEHOLD :0 HTTP

# Include another configuration file (Note that PrivateService of type HTTP
# will be rejected in that file and all its descendants)
%include /home/someuser/servrc.cfg.someuser

# Define PrivateService here as a real service
PrivateService HTTP /cgi-bin/privsrv.cgi?arg=val R=2000.0

# end of configuration
```

NCBI Intranet users **only** can take a look at a real configuration file on the host *ray* by clicking here.

Load-Balancing Service Mapping Client (`LBSMC`)

This is a special program, which repeatedly dumps onto the screen a table representing current contents of shared memory segment, where the daemon collects all information about hosts and services. The client's output can be controlled by a number of switches, full list of which can be obtained via switch *-h* alone, or NCBI Intranet users can click here. For example, to print only the hosts currently running the daemon one can use the following command:

```
>./lbsmc -s none 0
08/23/01 11:36:35 ================== sampson ===================================
Hostname/IPaddr Task/CPU LoadAv LoadBl   Status   StatBl n1 n2 n3 n4 n5 q1 q2 q3
iblastd            4/2     2.01   2.00   220.26     0.00  0  0  0  0  0 10 10  0
muncher        *  11/16    0.01   0.05 14414.42 15950.00  0  0  0  0  0 22 22  0
ray               29/6     2.71   3.14    75.39  2860.00  0  0  0  0  0  1  1  0
sampson           11/6     0.16   0.17  2173.91  5830.00  0  0  0  0  0  0  0  0
schroeder         60/8     0.45   0.00   285.72  8000.00  0  0  0  0  0  0  0  0
--------------------------------------------------------------------------------
* Hosts: 5/5, Svcs: 0/12/0      |      Heap: 8192, used: 1736/1776, free: 6416 *


08/23/01 11:36:35 LBSMD PID was detected as 9733
```

For NCBI Intranet users **only** the following live lists are available for browsing:

- Currently running hosts;
- Currently configured services and servers;
- Currently dead servers.

Some explanation to the client's output. The output usually contains 2 parts: first comes the host table, followed by the service table. If *-f* switch was specified, then the host table is prepended by raw heap printout, which shows the data exactly in the order they appear in the shared memory. The output goes in either long or short format depending on whether *-w* was specified on the command line (the switch requests the long (wide) output). Wide output occupies about 130 columns, while the short (normal) output takes 80 - the standard terminal size. In cases when the service name is more than the allowed number of characters to display, the trailing character will be shown as '>'. When there is more information to display about host/service, a '+' characters is set beside the host/service name (this additional information can be requested by switch *-i*). When both '+' and '>' are to be shown, they are replaced with '*'. In the wide output format, '#' shown in the service line tells that there is no host information available for the service (like for static servers). '!' character in the service line denotes that the service was configured/ stored with an error (this character actually should never appear in the listings, and should be reported whenever encountered). Wide output for hosts contain the time of bootup and startup. If the startup time is preceded by a tilde '~', then the host was gone for a while and then came back, while the client was running. A plus char '+' in the times is to show that the date belongs to the past year(s).

Server Penalizer

There is a means for application to report problems of accessing a certain server back to the load-balancing daemon, in the form of the penalty, a value in the range *[0..100]*, showing in percents how bad the server is. The value 0 means the server is completely okay, while 100 means the server (is misbehaving and) should **not** be used at all. The penalty is not a constant value: once set it starts to decrease in time first slowly, then faster and faster until reaches zero. This way, if a server was penalized for some reason beyond its control but afterwards the reason has gone, then the server becomes gradually available as its penalty (not being reset by applications again in the absence of the offending reason) goes itself to zero.

Technically, the penalty is maintained by a daemon, which has the server configured. That is, received by a certain host, which may be different from the one where the server was put into configuration file, the penalty first migrates to that host, and then the daemon on that host announces that the server was penalized. **Note**: Once the daemon is restarted, the penalty information is lost.

Service mapping API has a call ***SERV_Penalize()*** declared in *connect/ncbi_service.h*, which can be used to set the penalty for the last server obtained from the mapping iterator.

For script files (like ones used to start/stop servers) there is a dedicated utility program called `lbsm_penalize`, which sets penalty from command line. Because of intervening the load-balancing mechanism substantially this command should be used with extreme care.

`lbsm_penalize` is now a part of LBSM set of tools installed on all hosts that run `LBSMD`. As explained above, penalizing means making a server less favorable for choosing by load-balancing mechanism. Because of the fact that the full penalty of 100% makes a server unavailable for clients at all, at the time when the server is about to shut down (restart) it is wise to increase the server penalty to the maximal value, thus excluding the server from the service mapping. (Otherwise, the load-balancing daemon might not immediately notice that the server is down and continue dispatching to that server.) Usually penalty takes at most 5 seconds to propagate to all participating network hosts. That is, before an actual server shutdown the following sequence of commands can be used:

```
> ~ncbiduse/Service/lbsm_penalize 'Servicename STANDALONE 100 host 120' >
sleep 5 >
shutdown the server
```

The effect of the above is to set the maximal penalty *100* for the service `Servicename` (of type ***STANDALONE***) running on host `host` for at least *120* seconds - note the last numeric value. (After 120 seconds the normal behavior of the penalty occurs, i.e. automatical decreasing in time if not reset again to a different value with optional hold time.) Default hold time equals 0. In order for penalty to be realized by all hosts of the subnet, `sleep 5` precedes the server shutdown. Please note single quotes surrounding penalty specification: they are required in a command shell because `lbsm_penalize` takes only one argument, the entire penalty specification.

As soon as the server is down, `LBSMD` detects it in the matter of several seconds (if not instructed otherwise via configuration file) and then never dispatches to the server until it is up. In some circumstances the following command may come handy:

```
> ~ncbiduse/Service/lbsm_penalize 'Servicename STANDALONE 0 host'
```

it resets the penalty to 0 (no penalty) and is useful when, e.g. as for the previous example, the server is restarted and ready in less than 120 seconds but the penalty is still held high by `LBSMD`.

Network Service Mapper / Dispatcher (**`DISPD.CGI`**)

Note: Due to security issues, not all links in the public version of this file could be accessible by outside NCBI users. Unrestricted version of this document is available to inside NCBI users at: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/CPP_DOC/tools/dispatcher/DISPD.html.

The following addutional topics are discussed in this section:

- Overview
- Protocol Description
- Communication Schemes
- Server Launcher (NCBID.CGI)

Overview

`DISPD.CGI` is a CGI/1.0-compliant program, the purpose of which is to map requested service name to an actual server location, when the client has no direct access to `LBSMD`. This mapping is called *dispatching*. Optionally `DISPD.CGI` can also pass data between the client, requested the mapping, and the server, which implements the service, and was found as a result of *dispatching*. This combined mode is called *connection*. The client may choose either of these modes if there is no special requirement on data transfer (e.g. firewall connection). In some cases, however, the requested connection mode implicitly limits the request to be a *dispatching*-only request, and actual data flow between the client and the server occurs separately at a later stage.

Protocol Description

Dispatching protocol is designed as an extension to HTTP/1.0 and is coded in the HTTP header parts of the packets. The request (both *dispatching* and *connection*) is done by sending an HTTP-packet to `DISPD.CGI` with a query line of the form:

```
dispd.cgi?service=<name>
```

which can be followed by parameters (if applicable) to be passed to the service. `<name>` defines the name of the service to be used. The other parameters take form of one or more of the following construct:

```
&<param>[=<value>]
```

where square brackets are used to denote an optional value part of the parameter.

In case of a *connection* request, the request body can contain data to be passed to the first found server, connection to which is automatically initiated by `DISPD.GCI` after *dispatching*. On a contrary, in case of a *dispatching*-only request the body is completely ignored, that is the connection is dropped after the header has been read and then the reply gets generated without consuming the body data and that may confuse an unprepared client.

Mapping of service name into server address is done via LBSM Daemon (`LBSMD`), which has to run on the exactly same host where `DISPD.CGI` has started. `DISPD.CGI` never dispatches non-local client to a server marked as *local*-only (by means of *L=yes* in configuration of `LBSMD`). Otherwise, the result of dispatching is exactly what the client would get from service mapping API if run locally. Explicitly specifying its capabilities, the client can narrow the server search; for example, by choosing stateless servers only.

The following additional topics are discussed in this section:

- Client Request to DISPD.CGI
- DISPD.CGI Response to Client

Client Request to DISPD.CGI

In the client request to `DISPD.CGI` the following additional HTTP tags are recognized:

Accepted-Server-Types: <list> where `<list>` can include one or more of the following keywords separated by spaces:

- ***NCBID***
- ***STANDALONE***
- ***HTTP***
- ***HTTP_GET***
- ***HTTP_POST***
- ***FIREWALL*** which describe server types the client is capable to handle. Default is *any* (when the tag is not present at all in the HTTP header), and in case of *connection* request, the dispatcher will accomodate an actual found server with the connection mode, which the client requested, by relaying data appropriately, and in a way suitable for the server. **Note**: ***FIREWALL*** indicates that the client has chosen a firewall way of communication. **Note**: Some server types can be ignored if not compatible with current client mode.

Client-Mode: <client-mode> where `<client-mode>` can be one of the following:

- ***STATELESS_ONLY* -** specifies that the client is not capable of doing full-duplex data exchange with the server in a session mode (e.g. in a dedicated connection).
- ***STATEFUL_CAPABLE* -** should be used by the clients, which are capable of holding an open connection to a server. This keyword serves as a hint to dispatcher to try to open a direct TCP channel between the client and the server, thus reducing the network usage overhead.

Default (when the tag is not present at all) is ***STATELESS_ONLY*** in order to support Web-browsers.

Dispatch-Mode: <dispatch-mode> where `<dispatch-mode>` can be one of the following:

- ***INFORMATION_ONLY* -** specifies that the request is a *dispatching* request, and no data and/or connection establishment with the server required at this stage. That is, `DISPD.CGI` only returns a list of available server specifications (if any), corresponding to requested service, and in accordance with client mode and server acceptance.
- ***NO_INFORMATION*** - is used to disable sending the above mentioned dispatching information back to the client. This keyword is reserved solely for internal use by `DISPD.CGI` and should **not** be used by a side application.
- ***STATEFUL_INCLUSIVE* -** informs `DISPD.CGI` that current request is a *connection* request, and because it is going over HTTP, it is treated as stateless, thus the dispatching would supply stateless servers only. This keyword modifies the default behavior, and dispatching information sent back along with server reply (resulting from data exchange) should include stateful servers as well, so allowing the client to go to a dedicated connection later. **Note**: This keyword is not yet in use by present implementation of service connector.

Default (in the absence of this tag) is *connection* request, and as it is going over HTTP, it is automatically considered stateless. This is to support calls for NCBI services from Web-browsers.

Skip-Info-<n>: <server-info> A number of *<server-info>* strings can be passed to `DISPD.CGI` to ignore the servers from being potential mapping targets (in case the client knows the listed servers either do not work or are not appropriate). *Skip-Info* tags are enumerated by numerical consequent suffices (<n>), starting from 1. These tags are optional and should only be used if the client believes that the certain servers do not match the search criteria, or otherwise the client may end up with an unsuccessful mapping.

Client-Host: <host> This tag is used by `DISPD.CGI` internally to identify the `<host>`, where the request comes from, in case of relaying involved. Although `DISPD.CGI` effectively disregards this tag if the request originates from outside NCBI, and thus it cannot be easily fooled by the address spoofing, inhouse applications **should not** use this tag when connecting to `DISPD.CGI` because the tag **is trusted and considered** within the NCBI Intranet.

DISPD.CGI Response to Client

In response to the client `DISPD.CGI` can produce the following HTTP tags:

Relay-Path: <path> the tag shows how the information was passed along by `DISPD.CGI` and `NCBID.CGI`. This is essential for debugging purposes.

Server-Info-<n>: <server-info> the tag(s) (enumerated increasingly by suffix <n>, starting from 1) give a list of servers, where the requested service is available. The list can have up to 5 entries. However, there is only one entry generated when the service was requested either in firewall mode or by a Web-browser. For non-local client the returned server descriptors can include ***FIREWALL*** server specifications. Despite preserving information about host, port, type and other (but not all) parameters of the original servers, ***FIREWALL*** descriptors are not specifications of real servers, but they are created on-the-fly by `DISPD.CGI` to indicate that the connection point of the server cannot be otherwise reached without the use ofeither firewalling or relaying.

Connection-Info: <host> <port> <ticket> the tag gets generated in a response to a stateful-capable client and includes host (in a dotted notation) and port number (decimal value) of the connection point where the server is listening on (if either the server has specifically started or the firewall daemon has created that connection point due to the client request). The ticket value (hexadecimal) represents the 4-byte ticket that must be passed to the server as binary data in the very beginning of the stream. If instead of host, port and ticket information there is a keyword ***TRY_STATELESS***, then for some reason (see *Dispatcher-Failures:* tag below) the request failed, but may succeed if the client would switch into stateless mode.

Dispatcher-Failures: <failures> the tag value lists all transient failures, which dispatcher might have expirienced while processing request. Fatal error (if any) always appears as the last failure in the list. In this case, the reply body would contain a copy of the message, too. **Note**: Fatal dispatching failure is also indicated by an unsuccessful HTTP completion code.

Communication Schemes

After making *dispatching* request and using the dispatching information returned, the client can usually connect to the server on its own. Sometimes however, the client has to connect to `DISPD.CGI` again in order to proceed with communication with the server. For `DISPD.CGI` this would then be a *connection* request, which can go one of either 2 similar ways: *relaying* and *firewalling*.

- In *relay* mode `DISPD.CGI` passes data from the client to the server and back, playing the role of a middleman. Data *relaying* occurs when, for instance, a Web-browser client wants to communicate with a service governed by `DISPD.CGI` itself.
- In *firewall* mode `DISPD.CGI` only sends out the information about where the client has to connect to in order to communicate with the server. This connection point and verifiable ticket are specified in *Connection-Info:* tag in the reply header. **Note**: *Firewalling* is actually pertaining only to stateful-capable clients and servers.

Firewall mode is selected by the presence of keyword ***FIREWALL*** in *Accepted-Server-Types:* tag set by the client sitting behind a firewall, and not being able to connect to an arbitrary port.

These are scenarios of data flow between the client and the server depending on "stateness" of the client:

- *Stateless* client
    - Client is **not using firewall** mode:
        A. the client has to connect to the server by its own, using dispatching information obtained earlier, or

B. the client connects to `DISPD.CGI` in *connection* request (e.g. the case of Web-browsers), and `DISPD.CGI` makes data relaying for the client to the server.

- Client chooses to use *firewall* mode, then the only way to communicate with server is to connect to `DISPD.CGI` (making *connection* request), and use `DISPD.CGI` as a relay. **Note**: Even if the server is standalone (but *lackingS=yes* in configuration file of `LBSMD`), then `DISPD.CGI` initiates a microsession to the server and wraps its output into HTTP/1.0-compliant reply. Data from both `HTTP` and `NCBID` servers are simply relayed one-to-one.

- *Stateful-capable* client

    - Client **not using firewall** mode has to connect directly to the server, using dispatcher information obtained earlier (e.g. with the use of ***INFORMATION_ONLY*** in *Dispatch-Mode:* tag).

    - If *firewall* mode selected then the client has to expect *Connection-Info:* to come back from `DISPD.CGI` pointing where to connect to the server. If ***TRY_STATELESS*** comes out as a value of the former tag, then the client has to switch into stateless mode (e.g. by setting ***STATELESS_ONLY*** in *Client-Mode:* tag) in order for the request to succeed. **Note**: ***TRY_STATELESS*** could be induced by many reasons, mainly that all servers for the service are stateless ones, or that firewall daemon is not available on the host, where the client's request was received.

Note: Outlined scenarios show that no prior dispatching information is required for a stateless client to have in order to make a *connection* request, as `DISPD.CGI` can always be used as a data relay (this way Web-browsers can access NCBI services). But for stateful-capable client to establish a dedicated connection that additional step of obtaining dispatching information must precede the actual *connection*.

In order to support requests from Web-browsers, which are unaware of HTTP extensions comprising dispatching protocol, `DISPD.CGI` considers incoming request that does not contain input dispatching tags, as a *connection* request from a stateless-only client.

`DISPD.CGI` uses simple heuristics in analyzing HTTP header to determine whether the *connection* request comes from a Web-browser, or from an application (a service connector, for instance). In case of a Web-browser, the data path could be chosen more expensive but more robust, including connection retries if required, while on the contrary with an application, the dispatcher could return an error and the retry is delegated to the application.

`DISPD.CGI` always preserves original HTTP tags *User-Agent:* and *Client-Platform:* when doing both *relaying* and *firewalling*.

Server Launcher (NCBID.CGI)

There are servers of type NCBID, which are really programs that read requests from *stdin* and write responses into *stdout* without having sort of a common protocol. Thus, HTTP/1.0 was chosen as a framed protocol for wrapping both requests and replies, and NCBID.CGI utility CGI program was created to pass the request from HTTP body to the server and to put reply from the server into HTTP body and send back to the client. Also, NCBID.CGI is to provide a dedicated connection between the server and the client, if the client supports the stateful way of communication. Formely NCBID.CGI was implemented as a separate CGI program, but recently it was integrated into and became a part of DISPD.CGI (now NCBID.CGI is a symbolic link to DISPD.CGI).

NCBID.CGI determines the requested service from the query string the same way DISPD.CGI does so, i.e. by looking into the value of CGI parameter *service*. Executable file that has to be run is then obtained by searching configuration file (shared with LBSMD, default name is *servrc.cfg*): the path to the executable along with optional command-line parameters is specified after the bar character ("|") in the line containing the service definition.

NCBID.CGI can work in either of 2 connection modes, stateless and stateful, as determined by reading the following HTTP header tag: *Connection-Mode: <mode>* where <mode> is one of the following:

- ***STATEFUL***
- ***STATELESS***

Default (when the tag is missing) is ***STATELESS*** to support calls from Web-browsers.

When DISPD.CGI relays data to NCBID.CGI this tag is set in accordance with current client mode.

STATELESS mode is almost identical to a call of a conventional CGI program, except that in HTTP header there could be tags pertaining to dispatching protocol, and resulting from data relaying (if any) by DISPD.CGI.

In ***STATEFUL*** mode NCBID.CGI starts the program in a more tricky way, which is closer to work in firewall mode for DISPD.CGI. Namely, NCBID.CGI loads the program with its *stdin* and *stdout* bound to a port, which is made listening. That is the program becomes sort of an Internet daemon (only exception that exactly one incoming connection is allowed). Then the client is sent back an HTTP reply containing *Connection-Info:* tag. The client has to use port, host and ticket from that tag in order to connect to the server by creating a dedicated TCP connection. **Note**: NCBID.CGI**never** generates ***TRY_STATELESS*** keyword.

For the sake of backward compatibility, NCBID.CGI creates the following environment variables (in addition to CGI/1.0 environment variables created by the HTTP daemon when calling NCBID.CGI) before starting the service executables: NI_CLIENT_IPADDR and NI_CLIENT_PLATFORM. The former contains an IP address of the remote host (could be IP address of the firewall daemon if `NCBID.CGI` was started as a result of *firewalling*. The latter environment variable contains the client platform extracted from the HTTP tag *Client-Platform:* if any provided by the client.

NCBI Firewall Daemon (**`FWDaemon`**)

So-called "NCBI Firewall Daemon" (FD) is essentially a network multiplexer listening at an advertised network address.

FD works in a close cooperation with the NCBI network dispatcher, which informs FD on how to connect to the "real" NCBI server, and then instructs the network client to connect to FD (instead of the "real" NCBI server). Thus, FD serves as a middleman that just pumps the network traffic from the network client to the NCBI server and back.

FD allows network client to establish a persistent TCP/IP connection to any of publicly advertised NCBI services, provided that the client is allowed to make outgoing network connection to any of the following FD addresses (on front-end NCBI machines):

```
130.14.22.31, port 5812
130.14.22.32, port 5811
130.14.29.112, ports 5860..5870
```

NOTE: One FD can simultaneously serve many client/server pairs.

The following additional topics are discussed in this section:

- Using FD to connect from behind a "regular" firewall
- Using FD to connect from behind a "non-transparent" firewall
- Troubleshooting

Using FD to connect from behind a "regular" firewall

If your network client is behind a regular firewall, then just ask your system administrator to open the above addresses (only!) for outgoing connections, then set your client to "firewall" mode... and that's it, now your network client can use NCBI network services in a usual way (as if there were no firewall at all).

Using FD to connect from behind a "non-transparent" firewall

NOTE: If your firewall is "non-transparent" (it is extremely rare case), then your system administrator must "map" the corresponding ports on your firewall server to the advertised FD addresses (shown above). In this case, you will have to specify the address of your firewall server in the client configuration (CONN_PROXY_HOST).

The mapping on your non-transparent firewall server should look like this:

```
CONN_PROXY_HOST:5812       --> 130.14.22.31:5812
CONN_PROXY_HOST:5811       --> 130.14.22.32:5811
CONN_PROXY_HOST:5860..5870 --> 130.14.29.112:5860..5870
```

Please not that there is a port range that might not be presently used by any clients and servers, but is reserved for future extensions. Nevertheless, we recommend that you have this range configured on firewalls already now to allow the applications to function seamlessly in the future.

Troubleshooting

You can test if the FD ports are accessible from your host by just running, for example:

```
telnet 130.14.22.31 5812
```

and entering a line of arbitrary text in the TELNET session. If everything is fine, your TELNET session will look as follows (the line "test" is your input here):

```
> telnet 130.14.22.31 5812
Trying 130.14.22.31...
Connected to 130.14.22.31.
Escape character is '^]'.
test
NCBI Firewall Daemon: Invalid ticket. Connection closed.
See http://www.ncbi.nlm.nih.gov/cpp/network/firewall.html.
Connection closed by foreign host.
```

There is also an auxiliary UNIX shell script ***fwd_check.sh*** to check the *accessibility* of all of the above FD addresses.

The following dynamic Web page checks whether the FD is **running** on all hosts mentioned above.

NCBI Genome Workbench

NCBI Genome Workbench is an integrated sequence visualization and analysis platform. This application runs on Windows, Unix and Macintosh OS X.

The following topics are discussed in this section:

- Design Goals
- Design

Design Goals

The primary goal of Genome Workbench is to provide a flexible platform for development of new analytic and visualization techniques. To this end, the application must facilitate easy modification and extension. In addition, we place a large emphasis on cross-platform development, and Genome Workbench should function and appear identically on all supported platforms.

Design

The basic design of Genome Workbench follows a modified Model-View-Controller (MVC) architecture. The MVC paradigm provides a clean separation between the data being dealt with (the model), the user's perception of this data (provided in views), and the user's interaction with this data (implemented in controllers). For Genome Workbench, as with many other implementations of the MVC architecture, view and controller are generally combined.

Central to the framework is the notion of the data being modeled. The model here encompasses the NCBI data model, with particular emphasis on sequences and annotations. The Genome Workbench framework provides a central repository for all managed data, through the static class interface in CDocManager. CDocManager owns the single instance of the C++ Object Manager that is maintained by the application. CDocManager marshalls individual CDocument classes to deal with data as the user requests. CDocument, at its core, wraps a CScope class and thus provides a hook to the object manager.

The View/Controller aspect of the architecture is implemented through the abstract class CView. Each CView class is bound to a single document. Each CView class, in turn, represents a view of some portion of the data model or a derived object related to the document. This definition is intentionally vague; for example, when viewing a document that represents a sequence alignment, a sequence in that alignment may not be contained in the document itself but it is distinctly related to the alignment and can be presented in the context of the document. In general, the views that use the framework will define a top-level FLTK window; however, a view could be defined to be a CGI context such that its graphical component is a web browser.

To permit maximal extensibility, the framework delegates much of the function of creating and presenting views and analyses to a series of plugins. In fact, most of the basic components of the application itself are implemented as plugins. The Genome Workbench framework defines three classes of plugins - data loaders, views, and algorithms. Technically, a plugin is simply a shared library defining a standard entry point. These libraries are loaded on demand; the entry point returns a list of plugin factories, which are responsible for creating the actual plugin instances.

Cross-platform graphical development presents many challenges to proper encapsulation. To alleviate a lot of the difficulties seen with such development, we use a cross-platform GUI toolkit (FLTK) in combination with OpenGL for graphical development.